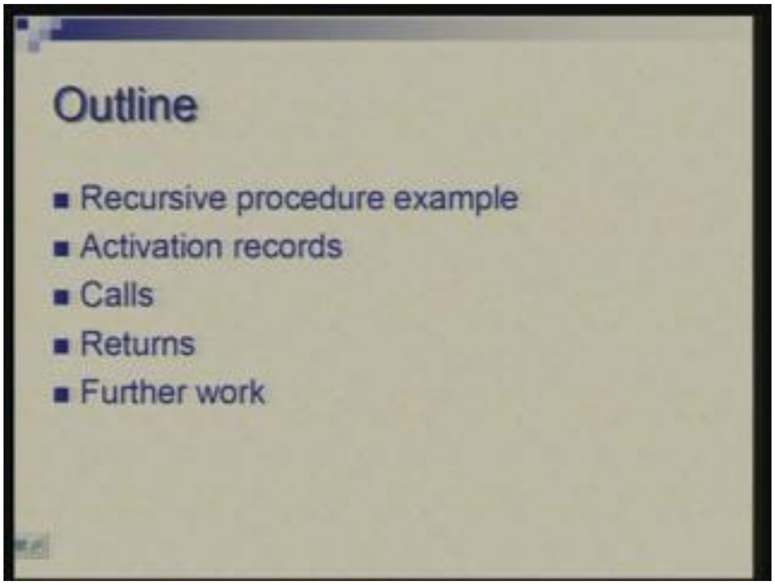
Computer Architecture
Prof. Anshul Kumar
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
Lecture - 6
Recursive Programs

In the previous class I discussed how subroutines or procedures can be defined and used in assembly language programs. We saw the use of instruction like jal and jr which are used to link the flow of control. We also talked about convention of register usage which helps in usage of temporary areas temporary words and also parameter passing. When it comes to recursive procedures you have to make use of a structure like stack where the information which goes in first comes out last and that is to match the order of calls and returns in the recursive procedures. So what we have seen that you need to define an activation record which is basically just a layout of memory area in stack how you are going to keep the temporary values, how you are going to keep the parameters and where you keep the local data.

Today I will take an example and try to elaborate on these ideas and show and illustrate how we can actually create activation records in order to define recursive procedures. Then we will see how at the time of function call these records get created and how they can be disposed off when you return from functions.

(Refer Slide Time: 2:17)

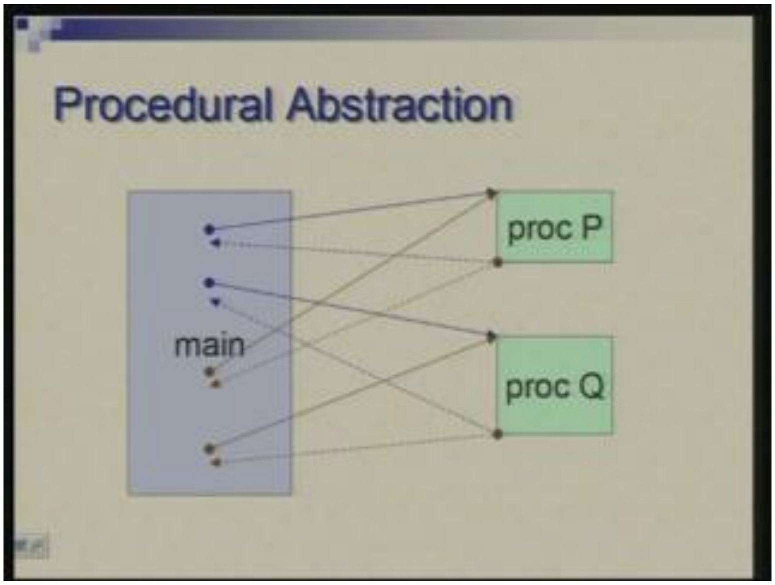


Outline

- Recursive procedure example
- Activation records
- Calls
- Returns
- Further work

I will actually do part of the examples and rest of the work would be carried out by you in your lab exercise.

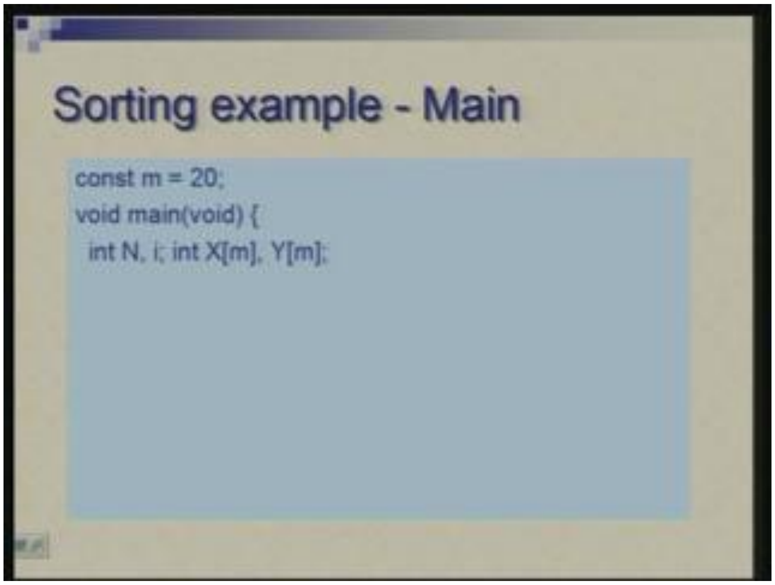
(Refer Slide Time: 2:44)



Procedural Abstraction

Recalling this picture that you have a main function which would call one or more functions or procedures and these calls could occur at several times. You have to arrange for flow of control as well as flow of data between these procedures and the main. I take an example of sorting procedure it is a merge sort you are familiar with; it can be implemented in a recursive manner or non-recursive manner. So just to illustrate the point I will take recursive implementation; first describe it in language like C and then we take up some crucial points in that implementation and see their assembly implementation.

(Refer Slide Time: 3:33)

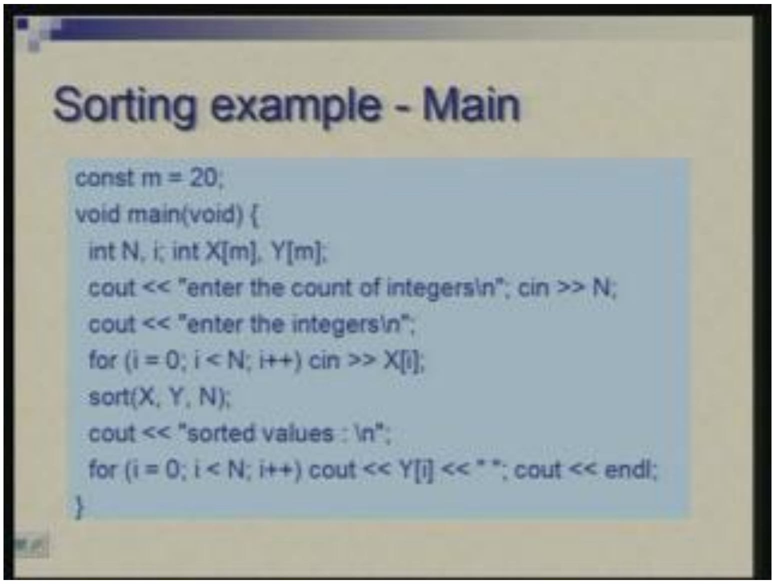


Sorting example - Main

```
const m = 20;
void main(void) {
  int N, i; int X[m], Y[m];
```

So we will begin with description of the main function which will essentially take care of inputting an array of integers then call the sort function and then output. The sort function would be a recursive function. So X is the array X is the array of dimension m, m has been defined as some constant here this is an input (Refer Slide Time: 4:08), Y would be the output so m is actually the maximum dimension and in this specific case we will have the dimension as input and that will be in variable N so N is the actual size in a particular case. this takes care of entering the value of N which is dimension of the array or size of the matrix then we go on to input the values so we are assuming that these are integers and integers are input and stored in X then we make a call to sort so X, Y, N are three parameters here then finally you have to output the sorted values. So this is taking care of the output. It is a very simple and straightforward function. You input, perform a sort and then output.

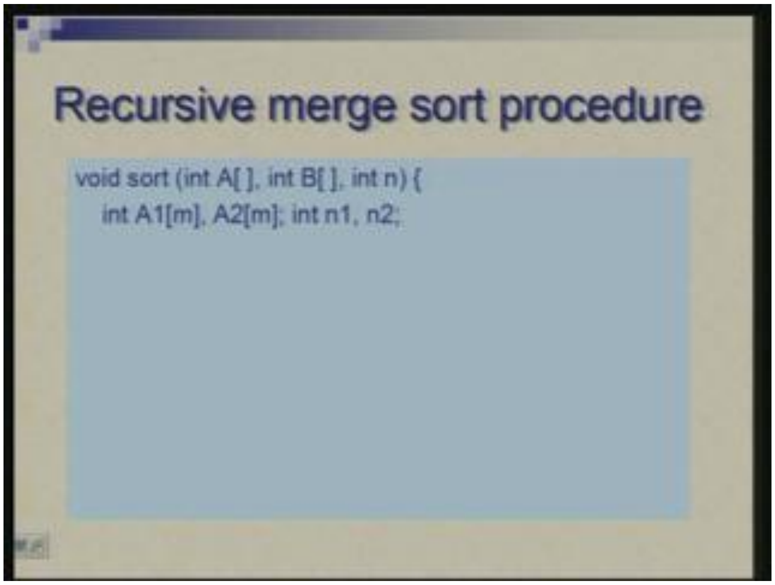
(Refer Slide Time: 5:10)



Sorting example - Main

```
const m = 20;
void main(void) {
  int N, i; int X[m], Y[m];
  cout << "enter the count of integers\n"; cin >> N;
  cout << "enter the integers\n";
  for (i = 0; i < N; i++) cin >> X[i];
  sort(X, Y, N);
  cout << "sorted values : \n";
  for (i = 0; i < N; i++) cout << Y[i] << " "; cout << endl;
}
```

(Refer Slide Time: 5:37)



Recursive merge sort procedure

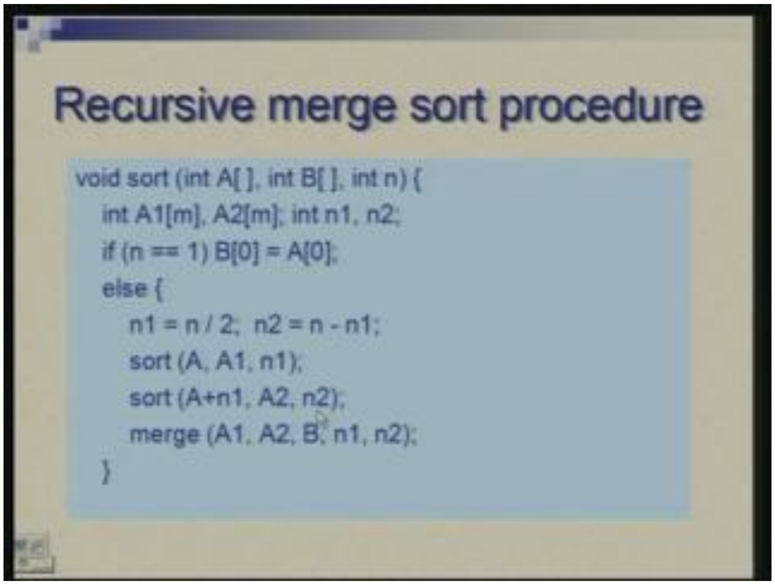
```
void sort (int A[ ], int B[ ], int n) {
    int A1[m], A2[m]; int n1, n2;
```

Let us look at the function. The merge sort function would be a recursive function. What we will do here is that use some local arrays in which we will keep the submatrices subarrays which are sorted and then merge them. So A1 and A2 are kept for this purpose. A special case is when n equal to 1 then you do not need to really split the array that array itself is sorted. When size is more than 1 then you split this into two parts, sort each of these and then merge the two that is the basic idea.

If n equal to 1 we simply copy from A to B else we find out two integers n1 and n2 n1 is roughly half of n and n2 is the remaining part. So the array A is going to be split into two halves roughly equal halves of size n1 and n2 and we sort them one by one. So we start with A so this is the input (Refer Slide Time: 6:57) that is the output, the output goes to a temporary array and the size of data is n1. The second one begins n1 words later the result goes to A2 and the size is n2.

Once you have done this then you simply need to merge the two. So the parameters are the two sorted subarrays A1 and A2, the result goes to B and the two sizes are n1 and n2. So, that is the end of the else part and that is the end of the function.

(Refer Slide Time: 7:31)



Recursive merge sort procedure

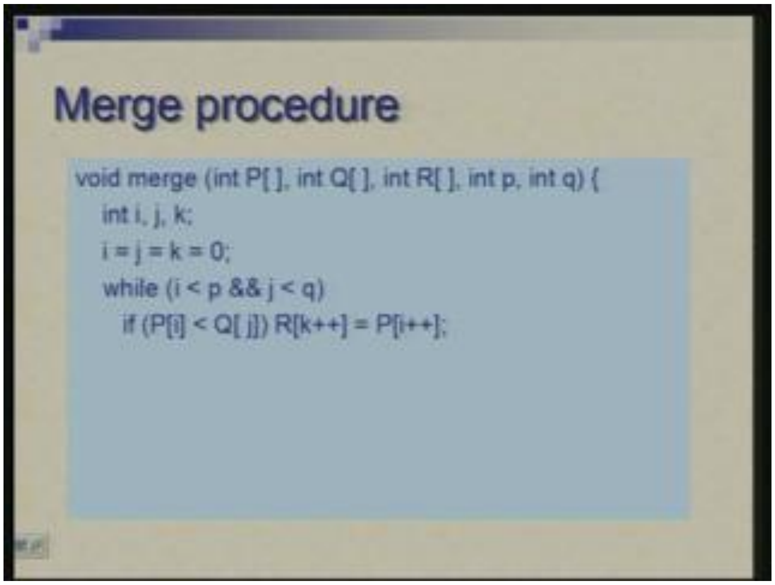
```
void sort (int A[ ], int B[ ], int n) {
   int A1[m], A2[m]; int n1, n2;
   if (n == 1) B[0] = A[0];
   else {
      n1 = n / 2;  n2 = n - n1;
      sort (A, A1, n1);
      sort (A+n1, A2, n2);
      merge (A1, A2, B, n1, n2);
   }
```

Its fairly straightforward in terms of its structure. One small thing which you should notice which is important here is that we have declarations of A1 and A2 here and the dimension is kept as m. C language will not permit a variable size array to be defined so we go by the maximum value which is possible.

As you keep on splitting the array the split portion will become smaller and smaller you would actually need less and less space but the way we have written we have to define somewhere else we have to go by the maximum possible size so there is a wastage of space but just for the sake of simplicity it has input in this particular manner. In fact I would like you to analyze how much space is getting utilized in this case. Every time you have a call to sort it creates two local arrays of size n.

Now, the function merge would take two arrays which have been sorted; simply merge them in such a manner that the order is maintained. So P and Q are the inputs, R is the output and small p small q gives the dimensions. These are i, j, k which are some local variables initialized to 0 these will act as the indices to the three arrays i, j and k will respectively index to p q and r. So here is the beginning of a while loop which will go over the arrays as long as both the arrays are not fully scanned. So i and j are indices to p and q and as long as these are less than the max limits you keep on scanning and try to find which is the smaller element which is the larger element and pick up the one which is smaller. So we are creating sorted array in ascending order.

(Refer Slide Time: 10:01)

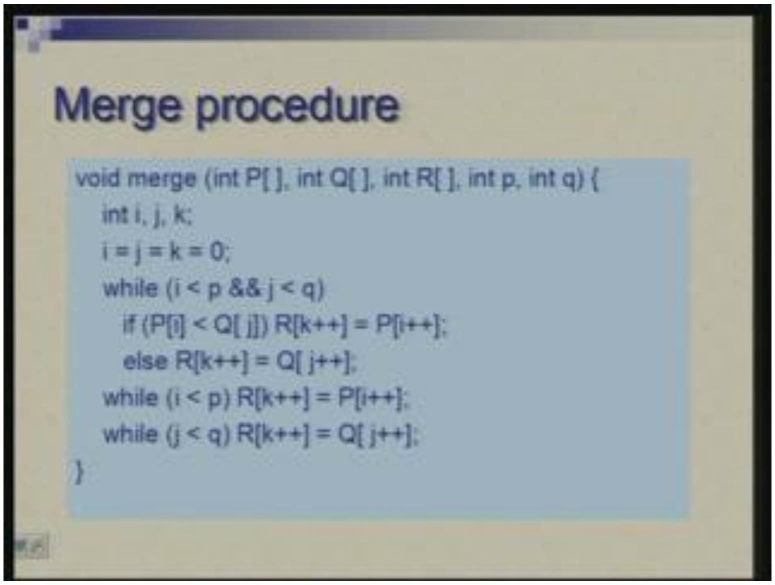


Merge procedure

```
void merge (int P[ ], int Q[ ], int R[ ], int p, int q) {
    int i, j, k;
    i = j = k = 0;
    while (i < p && j < q)
      if (P[i] < Q[ j]) R[k++] = P[i++];
```

So, if P[i] is less than Q j you transfer one element from P to R otherwise transfer one element from Q to R and also take care of updation of the indices. Then you reach a point where possibly you have exhausted one of the arrays or may be both the arrays. you need to see if there are remaining elements either p or q they need to be passed on to R as they are, if they are already in order and they go at the tail end of R. So, if i has not reached p yet you need to pass on elements from P to R and if j has not reached q you need to pass on remaining elements from Q to R. this is the end of while.

(Refer Slide Time: 11:04)

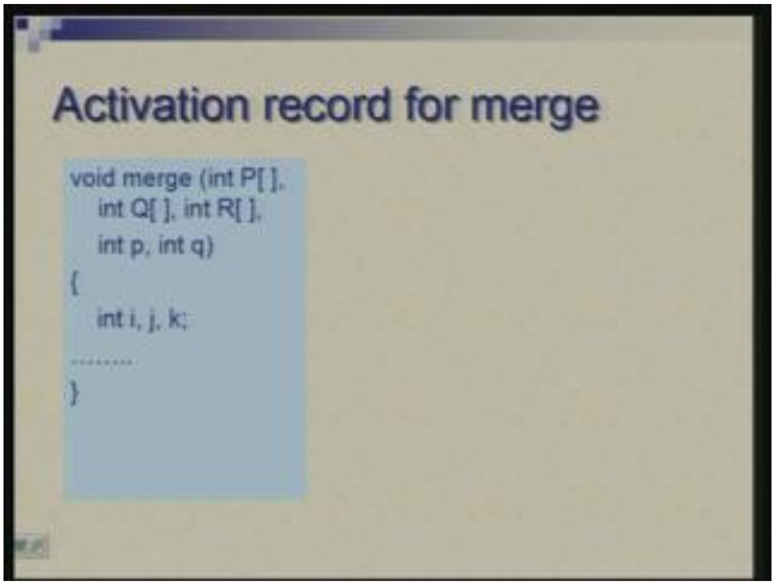


Merge procedure

```
void merge (int P[ ], int Q[ ], int R[ ], int p, int q) {
    int i, j, k;
    i = j = k = 0;
    while (i < p && j < q)
      if (P[i] < Q[ j]) R[k++] = P[i++];
      else R[k++] = Q[ j++];
    while (i < p) R[k++] = P[i++];
    while (j < q) R[k++] = Q[ j++];
}
```

Now we look at how we implement these recursive procedures in assembly and as I mentioned earlier that our implementation will be based on defining activation records creating them at the time of call and disposing them of at the time of return. so let us define the structure for these activation records.

(Refer Slide Time: 11:54)

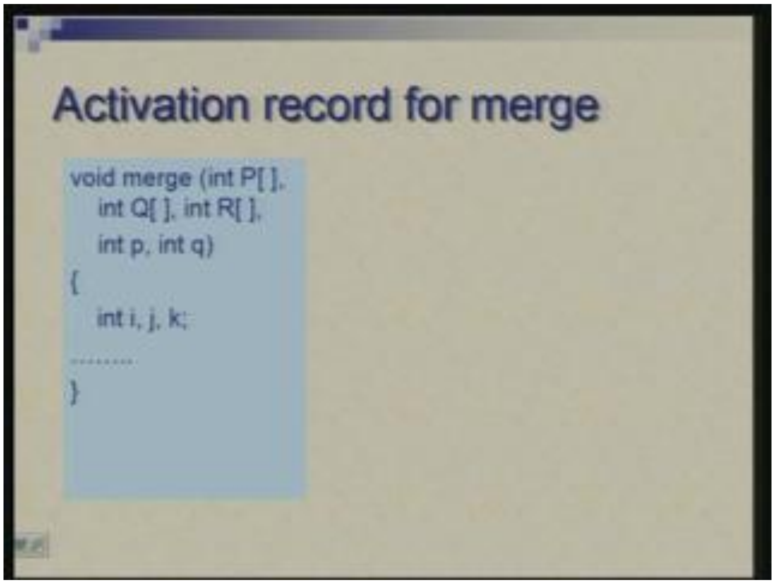


Activation record for merge

```
void merge (int P[ ],
    int Q[ ], int R[ ],
    int p, int q)
{
    int i, j, k;
.........
}
```

Therefore, first we look at the merge procedure. It has these parameters P, Q and R three arrays and two scalars p and q which give the dimensions. So accordingly we will create activation record, we also will need to take care of the local variables. We had three local variables i, j and k.

(Refer Slide Time: 12:04)



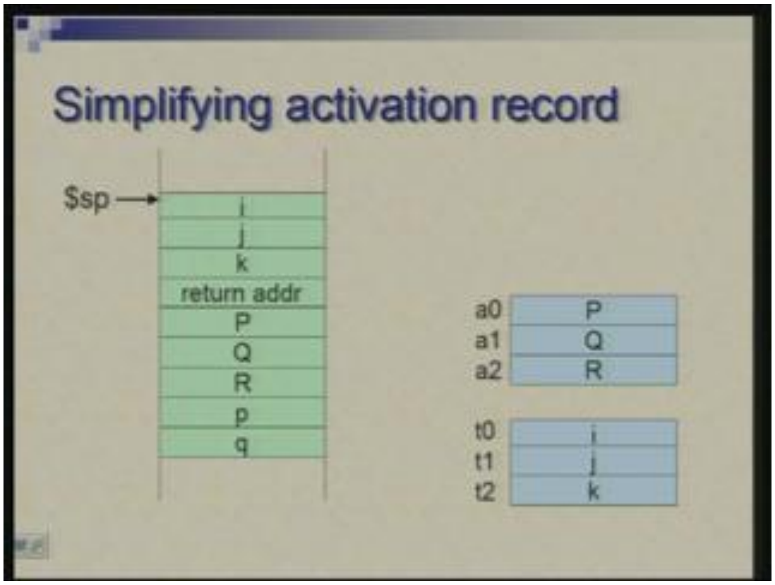
Activation record for merge

```
void merge (int P[ ],
    int Q[ ], int R[ ],
    int p, int q)
{
    int i, j, k;
.........
}
```

This is a possible activation record. What we have done is we have placed the parameters here, there is a return address and the local variables. the local variables are i, j and k parameters addresses, starting addresses of array P, Q and R and the values small p and small q. There is no sanctity of the order in which I have put things here the order could have been changed but I have just put in the order in which things are appearing.

What we are saying is that when this merge has to be called we would reserve space on top of the stack which will accommodate all these parameters, return address and the local data; at the time of return the stack would be shrunk and this area would be inaccessible. This is one way but we can simplify it further.

(Refer Slide Time: 13:21)



Simplifying activation record

$sp →

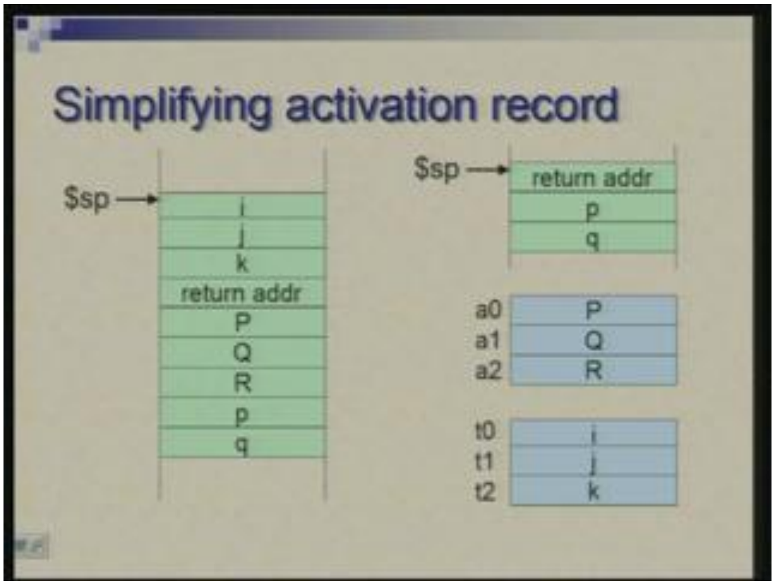
i
j
k
return addr
P
Q
R
p
q

a0	P
a1	Q
a2	R

t0	i
t1	j
t2	k

You recall that there are four registers a0 to a3 which are designated for passing parameters. So we can also use those registers for passing parameters and since merge is not being called recursively it is good enough to put up to four parameters in those registers. I have shown a possible way of creating activation record where P, Q and R addresses are put in register a0 a1 a2; we have two more so let us keep them in the stack and the local variables i, j and k can be kept in registers t0, t1, t2 etc which are temporary registers. With this we have a smaller activation record which needs to take care of return address and two of the parameters small p and small q.

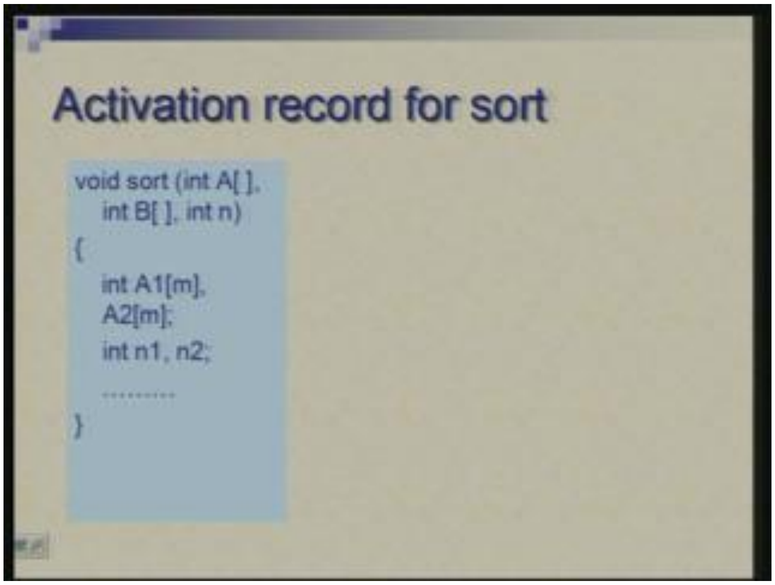
(Refer Slide Time: 14:30)



Simplifying activation record

Basically my entire requirement is now split between registers following the usual convention and something is still on the stack. you could arrange to go little bit beyond the convention for example, use some additional registers because we have five parameters here and convention is defined for only four as far as registers are concerned but I can put in some more registers if they are not required elsewhere to pass on all the parameters through registers. But let me show you a mix approach where something has put into registers and something is still in activation record.

(Refer Slide Time: 15:26)

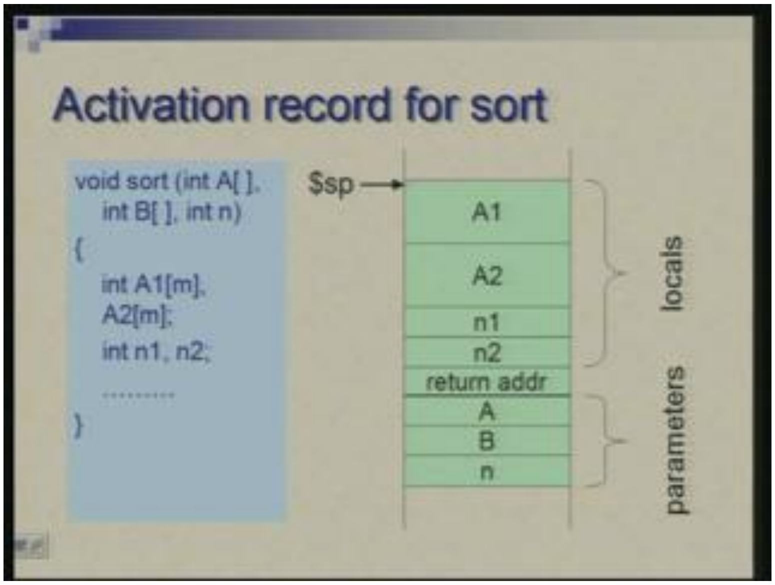


Activation record for sort

```
void sort (int A[ ],
    int B[ ], int n)
{
    int A1[m],
    A2[m];
    int n1, n2;
    ..........
}
```

For procedure sort this indicates the requirement. We have three parameters here two array addresses and one integer value and there is much more of local data here. There are two local arrays A1 and A2 and two scalars n1 and n2 so we will put all this in activation record.

(Refer Slide Time: 15:53)

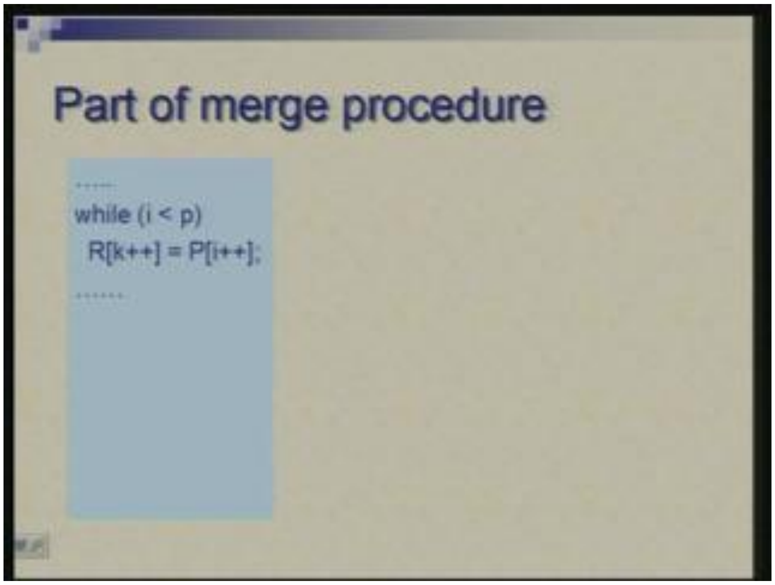


Activation record for sort

```
void sort (int A[ ],
    int B[ ], int n)
{
    int A1[m],
    A2[m];
    int n1, n2;
    .........
}
```

So at the bottom again following similar idea we keep the parameters. A address B address and n value these are here then there is space for return address. Now, out of the local data A1 A2 n1 and n2 we have put them in this order the entire array A1, the entire array A2, value n1, value n2 so these are here in upper part of the activation record. We are not initializing these locations it is only that you need to keep space so space has to be allocated here which only means that you simply move the stack pointer there and the function or the procedure will have to explicitly initialize any of this data if necessary.

(Refer Slide Time: 17:04)

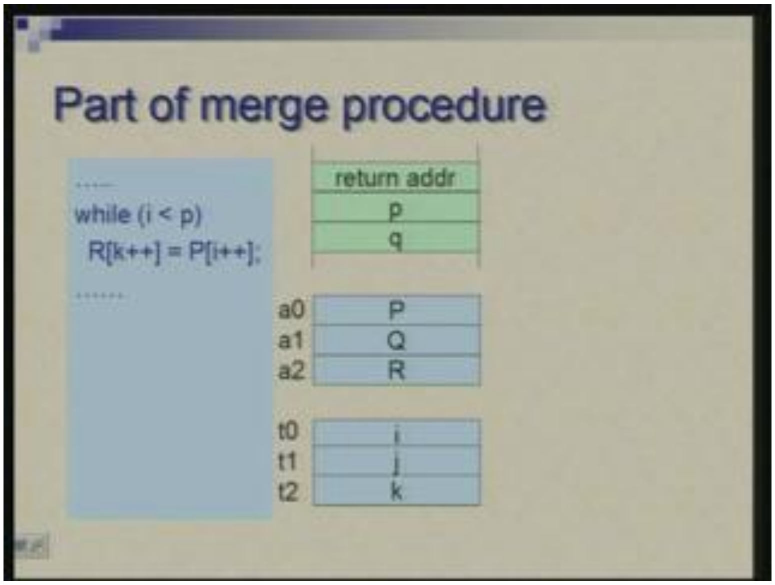


Part of merge procedure

```
......
while (i < p)
  R[k++] = P[i++];
.......
```

Now, having defined activation records we need to see how we use it, how we create it and how we use it. So, first let me show you the usage. Here I am showing (Refer Slide Time: 17:17) part of the code part of the code of merge function. You remember, after the main loop there was one loop where you were transferring elements from P to R after one of the arrays has been exhausted. So let us see how we will encode this part keeping in mind where our parameters are kept and where our local values are kept.

(Refer Slide Time: 17:41)



Part of merge procedure

```
......
while (i < p)
  R[k++] = P[i++];
.......
```

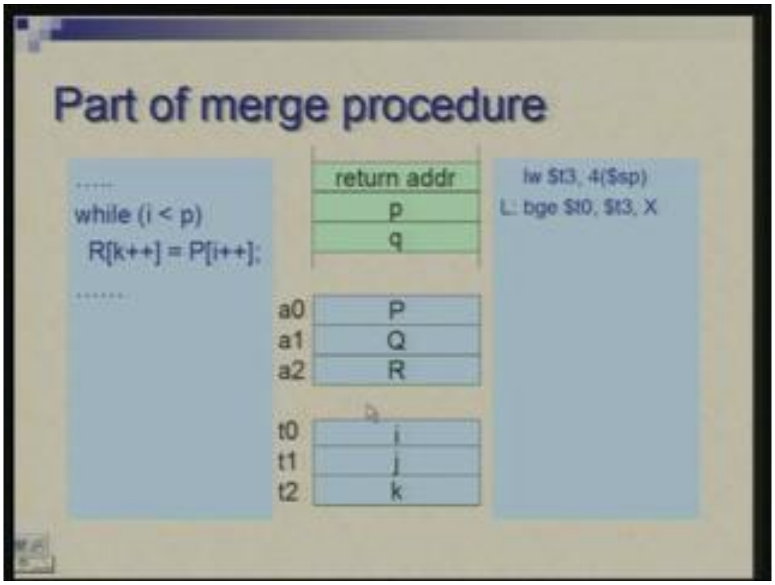
return addr
p
q

a0	P
a1	Q
a2	R

t0	i
t1	j
t2	k

I am going to follow the second approach where some of the parameters are passed through registers and some are passed through the stack and the location where i, j k the local variables are kept is in temporary registers. So we begin by accessing the value p which is kept in second location from top in the activation record. So, if you say load word within offset of 4 and register being sp……… basically your stack pointer is here pointing to top of the stack.

(Refer Slide Time: 19:09)



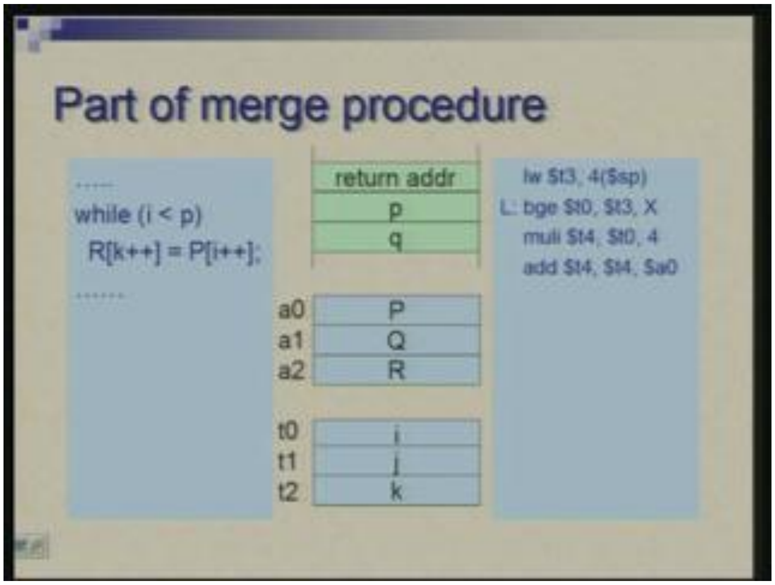
Part of merge procedure

```
......
while (i < p)
  R[k++] = P[i++];
......
```

```
  lw $t3, 4($sp)
L: bge $t0, $t3, X
```

The address of p is contents of stack pointer plus 4. Thus, you are basically loading p keeping in register t3 where it will be compared with i and i is located in t0 it is already available in a register so you can compare….. I am using a pseudo instruction bge branch if greater than equal to so if i is greater than equal to p then I branch of to a point X which will be somewhere towards the end of the program otherwise it begins loop so I need to prepare the indices for arrays R and P.

(Refer Slide Time: 19:50)



Part of merge procedure

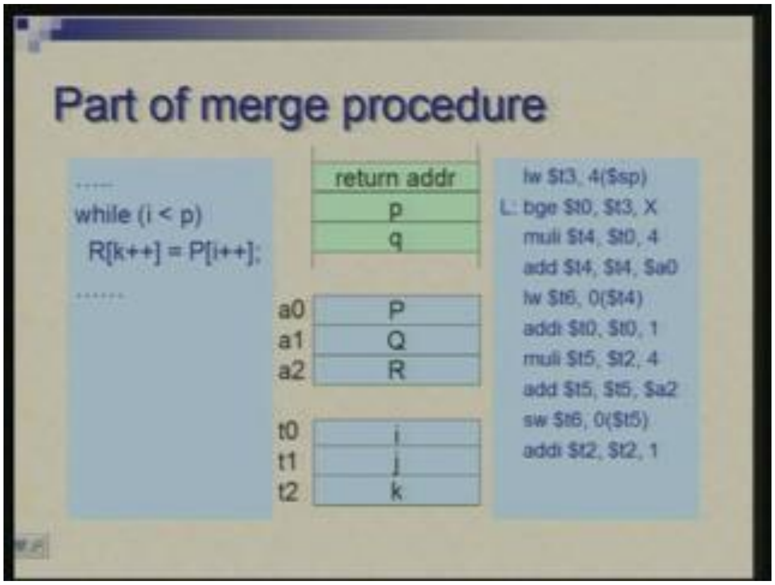
```
......
while (i < p)
  R[k++] = P[i++];
.......
```

a0	P
a1	Q
a2	R
t0	i
t1	j
t2	k

```
   lw $t3, 4($sp)
L: bge $t0, $t3, X
   muli $t4, $t0, 4
   add $t4, $t4, $a0
```

Therefore, first I take t0 which is i multiplied by 4 and add to the starting address of array P which is available in a0. Now after these two steps multiplication and addition t4 has the address of P[i] and we can bring one word from P[i] into a register t6 and we also take care of i plus plus that is t0 is incremented then we are preparing address for R[k] so k is available in t2 multiplied by 4 then add the starting address of R. now, t5 has the address of R[k] the value which was read from array P was in t6 here now it is stored back in array R, the address or the index k is incremented by 1 and that completes this particular loop.

(Refer Slide Time: 21:19)



Part of merge procedure

```
......
while (i < p)
  R[k++] = P[i++];
.......
```

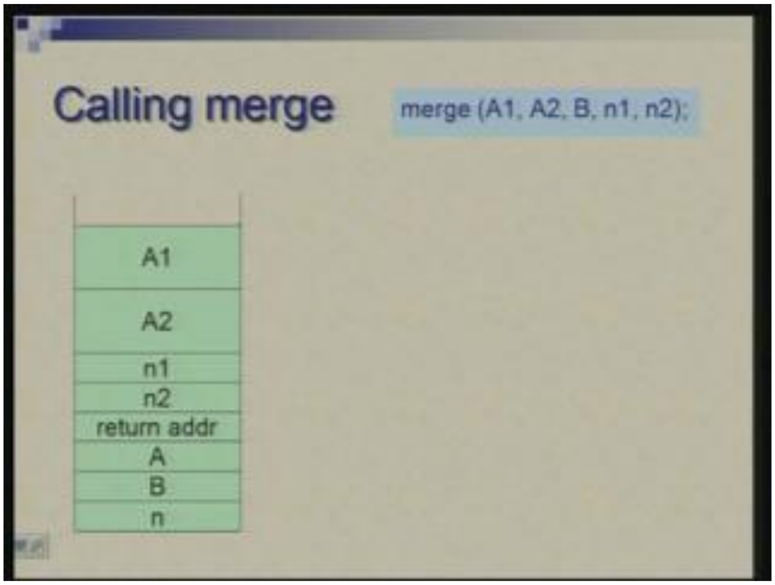
a0	P
a1	Q
a2	R
t0	i
t1	j
t2	k

```
   lw $t3, 4($sp)
L: bge $t0, $t3, X
   muli $t4, $t0, 4
   add $t4, $t4, $a0
   lw $t6, 0($t4)
   addi $t0, $t0, 1
   muli $t5, $t2, 4
   add $t5, $t5, $a2
   sw $t6, 0($t5)
   addi $t2, $t2, 1
```

Now, j L will take you back to the beginning of the loop and the label X is where the loop ends. So basically what I have tried to show here is that you keep this picture of activation record in front of you and you know what is lying where so accordingly you can pick the data either from registers or from activation records and whatever you are picking from activation record you can actually find out the offset required with respect to the stack pointer and access that value.

This was how we used once the activation record for merge that was created and you were inside the procedure merge. I have not described the entire procedure but just took one sample out of that. Now at the calling point when you are calling this function merge you need to create activation record and now we will focus on that.

(Refer Slide Time: 23:23)



Calling merge

merge (A1, A2, B, n1, n2);

A1

A2

n1

n2

return addr

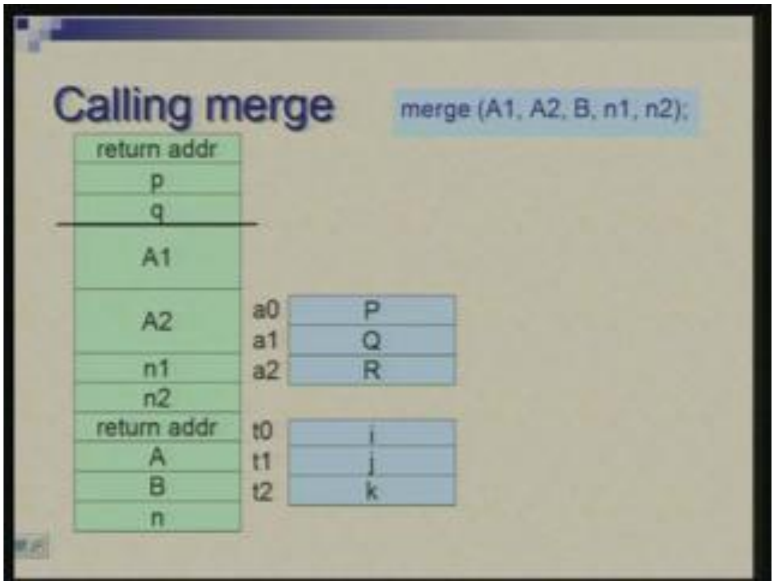
A

B

n

This is activation record of sort procedure. When a call is made the activation record of merge would be created on the top of this and the values we were trying to put in the registers will be put in the registers so this is the effect which we want to achieve that starting with this, this is where the stack is at the moment and execution of this call should result in creation of one additional record on top of the stack and filling up of the registers with appropriate values. So let us see how it is done.

(Refer Slide Time: 23:55)



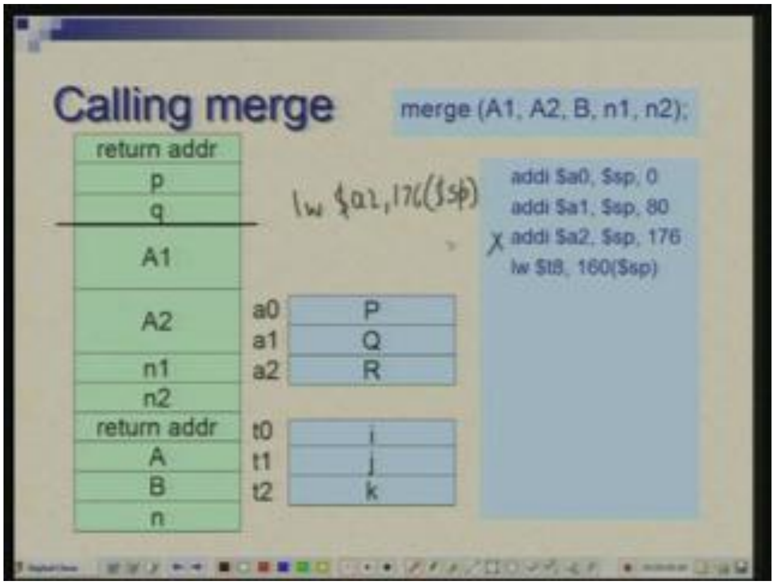
Calling merge

merge (A1, A2, B, n1, n2);

Here a0 is supposed to contain the address of p which is…….. now the value which is being passed the array which is being passed as an argument for p is a1, so a0 should contain actually address of starting address of a1 which is on top of the stack. So the register sp contains the address of starting address of a1 and that value is being moved into a0 so this statement takes care of filling up appropriate value in a0 as a parameter.

A1 should contain starting address of Q and the parameter being passed for Q is A2. So A2 is starting twenty words down after A1 or which means 80 bytes so sp plus 80 is the starting address of second array second parameter or Q which we are putting in A1. Then the third parameter which is to be kept in A2 is R and actual argument is B so now array B is not here but it is the address of B which is here and you can see how much below it is we have 20 plus 20 which is 40 41 42 43 44 so you leave a gap of 44 words or 176 bytes and you are at a place where you have the third argument. I think this is not correct….. I think the third statement requires correction…. let me… I have put this address of B there but we should put the contents of this location there so it should be……

(Refer Slide Time: 27:32)



Calling merge

merge (A1, A2, B, n1, n2);

```
addi $a0, $sp, 0
addi $a1, $sp, 80
addi $a2, $sp, 176
lw $t8, 160($sp)
```

lw $a2,176($sp)

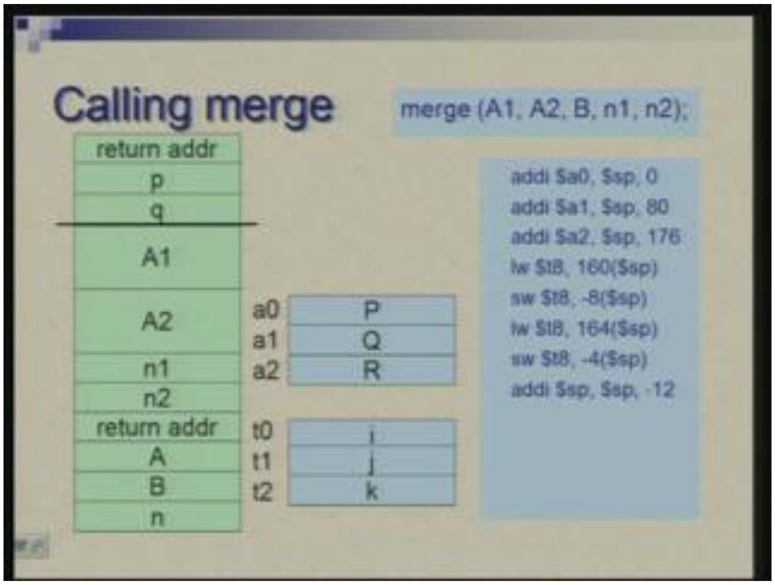
Please replace this with that (Refer Slide Time: 27: 32) .you are not storing the value sp plus 176 there you need to read that location and whatever is content there that address need to be put in A2.

The next parameter is n1 which we need to keep in the stack here and n2 needs to be kept in the stack in the activation record so these are available in the previous activation records. So from here we need to copy them. This is copying it….. first we bring it to some register t8 and n1 is at how much offset from the stack pointer; stack pointer is still here so 20 plus 20 40 words down from the top of the stack which means 160 bytes.

We read from offset of 160 from top of the stack it is brought to some temporary register t8 and where do we store it is at minus 8 offset from top of the stack. So top is still here (Refer Slide Time: 29:15) and we store it at two words above the top of the stack which means minus 8 offset. So look at it carefully. We have brought a value from plus 160 offset and stored it at minus 8 offset that means we have read it from here and kept it there.

Similarly, the next two steps next two instructions will bring the value of n2. So you take it from 164 offset bring it to minus 4 offset. So you have copied these two values to create the activation record and stack pointer is raised by 12 bytes. So basically we have created activation record of three words; two words we have already filled in and after jal instruction when you reach the beginning of the merge function then you will fill up the return address there.

(Refer Slide Time: 30:00)



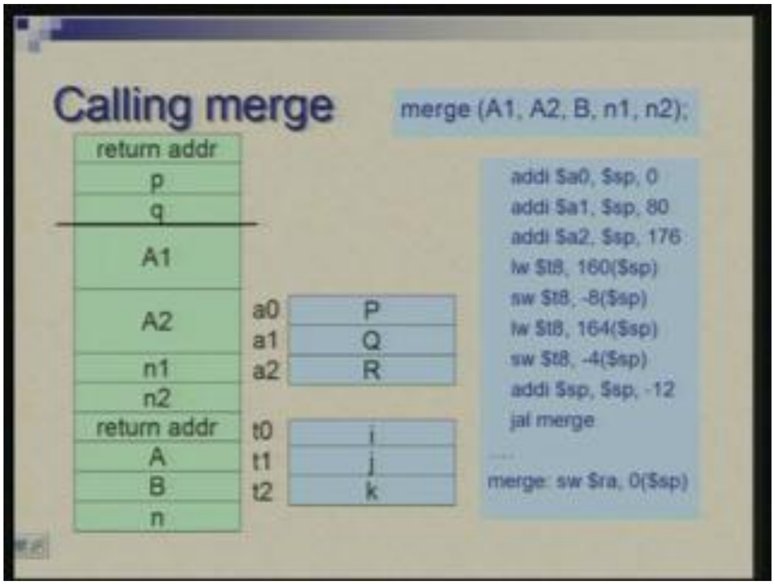
Calling merge

merge (A1, A2, B, n1, n2);

```
addi $a0, $sp, 0
addi $a1, $sp, 80
addi $a2, $sp, 176
lw $t8, 160($sp)
sw $t8, -8($sp)
lw $t8, 164($sp)
sw $t8, -4($sp)
addi $sp, $sp, -12
```

Now you make a call and the first instruction in the merge function would be saving the return address at top of the stack.

(Refer Slide Time: 30:27)



Calling merge

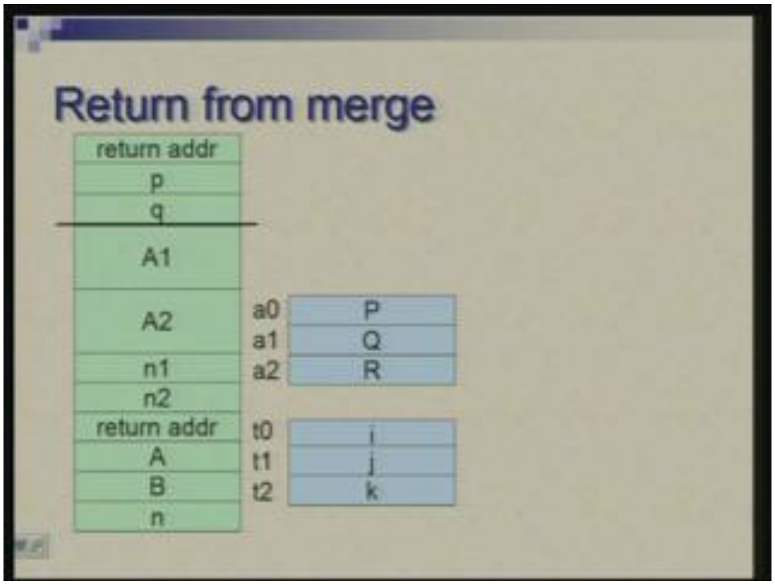
merge (A1, A2, B, n1, n2);

```
addi $a0, $sp, 0
addi $a1, $sp, 80
addi $a2, $sp, 176
lw $t8, 160($sp)
sw $t8, -8($sp)
lw $t8, 164($sp)
sw $t8, -4($sp)
addi $sp, $sp, -12
jal merge
.....
merge: sw $ra, 0($sp)
```

Therefore, this takes care of creating small activation record filling in the parameters which will be passed through registers into appropriate registers and when the call has occurred when the control has been transferred then you take the return address and put it in appropriate place in the stack. This is how the call would a simple single statement which you see like this would actually result in that much of assembly code.

Now let us see how you return from merge function. We will have to dispose of the activation record which is created. We had added one activation record on the top. I am again showing that at the top there is activation record for merge; at the bottom there is record for sort and registers are containing parameters and temporary data.

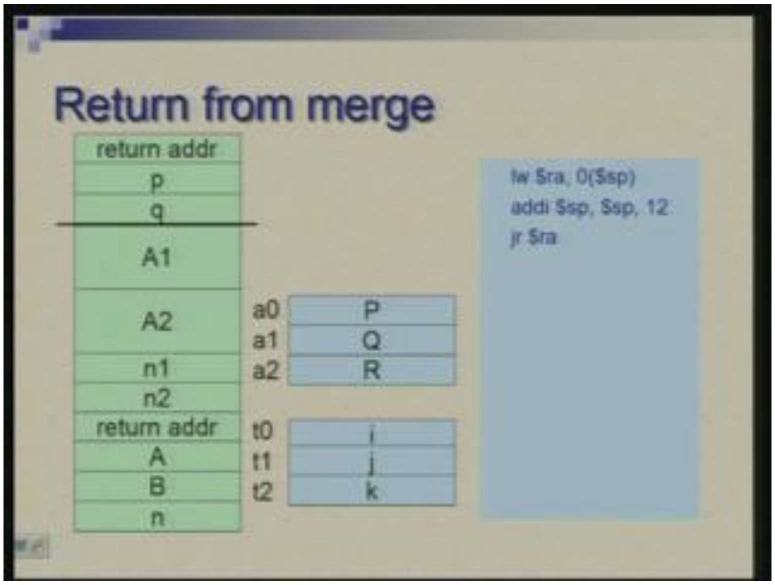
(Refer Slide Time: 31:47)



Return from merge

So, before you could return basically these three statements before returning you must bring the return address into ra and that we can simply do from top of the stack.

(Refer Slide Time: 32:19)

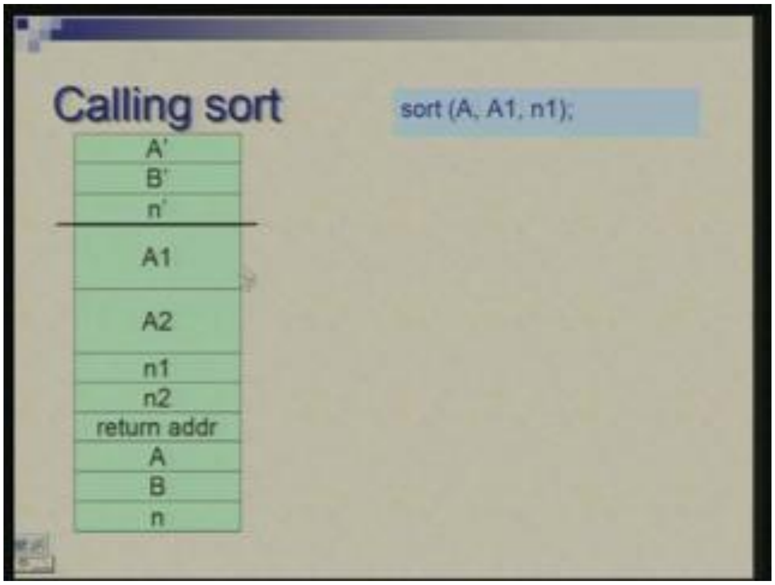


Return from merge

```
lw $ra, 0($sp)
addi $sp, $sp, 12
jr $ra
```

Now first load statement loads return address into ra; adding 12 to the stack pointer actually shrinks the stack and basically the activation record is disposed of. We do not have to fill in 0 but just bring the stack pointer down and this jr or jump register brings the control back so return is comparatively simpler.

(Refer Slide Time: 32:45)



Calling sort

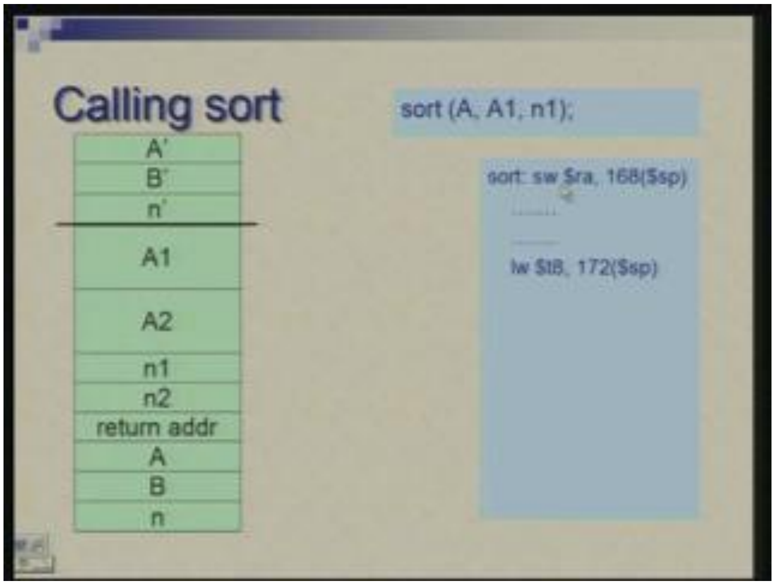
```
sort (A, A1, n1);
```

A'
B'
n'
A1
A2
n1
n2
return addr
A
B
n

Now we go to the sort function. Particularly we look at the point where sort is being called recursively from within the function. You would recall that there are two calls to sort and we will look at the first one of these. So once again I will imagine that there is a activation record which corresponds to the current invocation of the sort function. Over this we are going to build another activation record. I am showing only part of that which corresponds to the parameters. So, when you create a new record you will create a copy of the same thing copy of the same activation record but the value is filled in differently.

Now in this recursive call the parameters being passed are (A, A1 and n1). These three parameters which I am labeling as A prime B prime and n prime would get these values address of A address of A1 and the value n1 and let us see how this is done.

(Refer Slide Time: 34:27)



Calling sort

A'
B'
n'
A1
A2
n1
n2
return addr
A
B
n

```
sort (A, A1, n1);
```

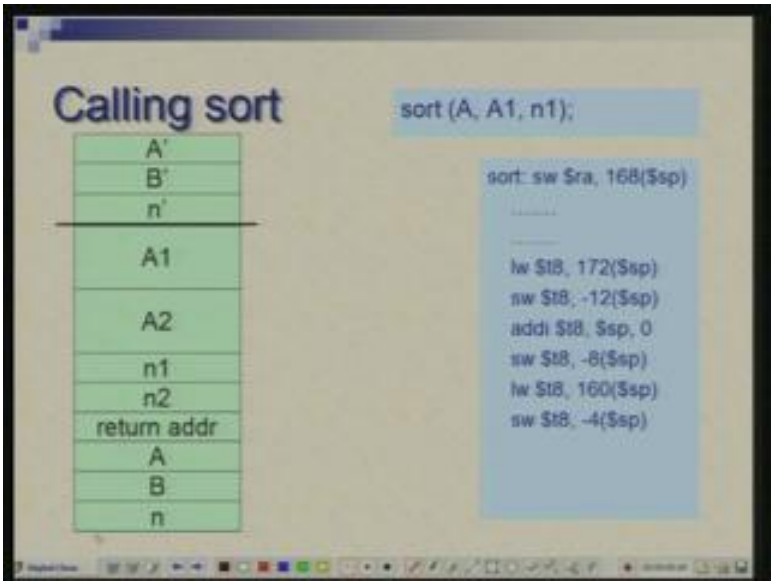
```
sort: sw $ra, 168($sp)
      ........
      ________
      lw $t8, 172($sp)
```

I am showing the first statement of sort function which will save the return address. Return address is being saved in a location which is 168 offset from top of the stack so 20 plus 20 is 40 41 42 and 43 sorry after 42 you have to store so 42 times four is the byte offset which is 168 so store return address at location 168 deep in the stack.

Now, I am coming to the point which is corresponding to this sort call. Here I have to take this address place it here; the address of A1 place it at the second position and the value n1 place at the third position. So these two addresses and one value have to be kept. first let us pick up the address of A which is lying here; so it is the address which is here we need to take this is at offset 20 plus 20 40 41 42 43 and 43 into 4 is 172 so at that offset we load the address of A into t8 and store it at minus 12 offset which means the top position of this picture.

Next we look at the contents of stack pointer which is pointing to top of the activation record which means the address of A1 that is brought to t8 and stored in minus 8 offset that is this position. Then we look at value n1 which is at 160 offset and stored at minus 4 offset. Let me put down these offsets so that you can see things very clearly.

(Refer Slide Time: 36:38)



Calling sort

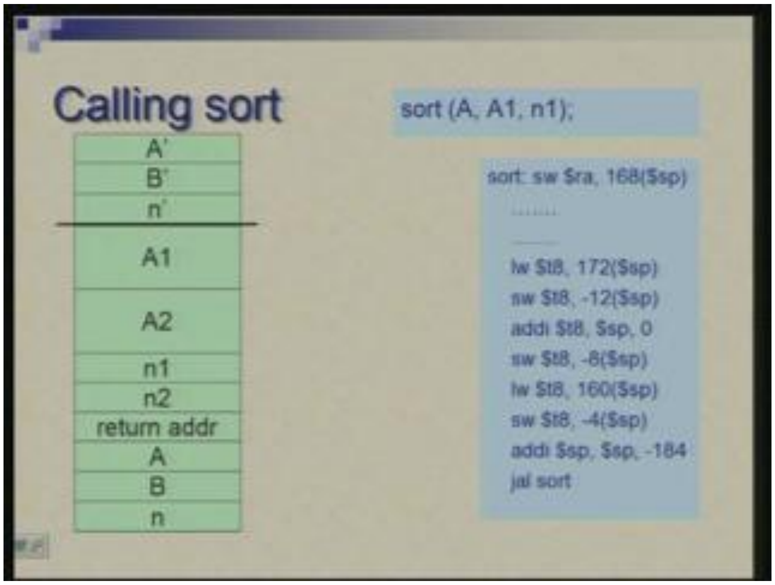
A'
B'
n'
A1
A2
n1
n2
return addr
A
B
n

sort (A, A1, n1);

```
sort: sw $ra, 168($sp)
   .........
   .........
   lw $t8, 172($sp)
   sw $t8, -12($sp)
   addi $t8, $sp, 0
   sw $t8, -8($sp)
   lw $t8, 160($sp)
   sw $t8, -4($sp)
```

This is offset 0, this is offset 80, this is 160, 164, 168, 172, 176 (Refer Slide Time: 37:05) we do not need these immediately…. 180 and this is minus 4 minus 8 and minus 12. Now you can easily correlate; we have brought the address from 172 placed it in minus 12 we have taken this address in stack pointer itself placed it in minus 8, we brought the value from 160 offset placed it in minus 4. So now the activation record generation is not complete yet. We have taken care of the parameters but we need to create space for local data and all you need to do is find out what is the total space required and just push the stack pointer by that much value. So it is 184, 184 is the total size; you can tell that two arrays twenty each that is 40 so 1 2 3 4 5 6 so total of forty six words or 184 bytes. So, after you create the space simply transfer the control to appropriate locations.

(Refer Slide Time: 38:47)



Calling sort

A'
B'
n'
A1
A2
n1
n2
return addr
A
B
n

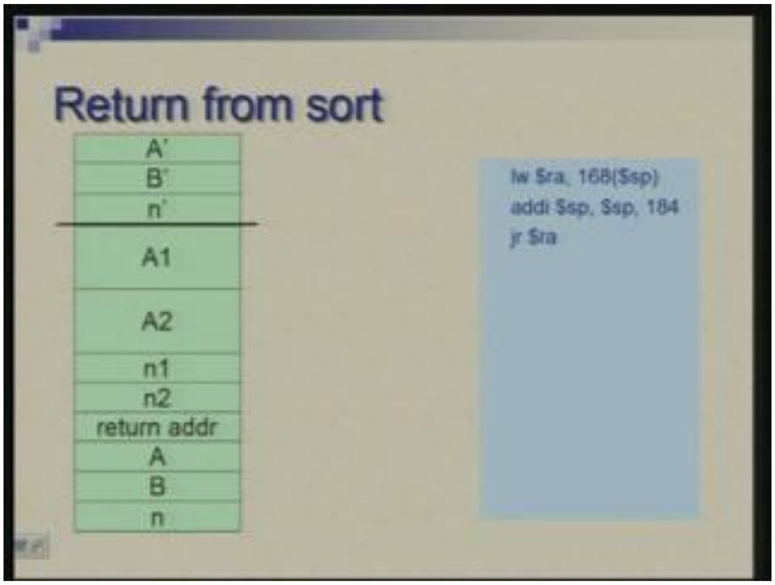
```
sort (A, A1, n1);
```

```
sort: sw $ra, 168($sp)
    ........

    ........
    lw $t8, 172($sp)
    sw $t8, -12($sp)
    addi $t8, $sp, 0
    sw $t8, -8($sp)
    lw $t8, 160($sp)
    sw $t8, -4($sp)
    addi $sp, $sp, -184
    jal sort
```

How do you return from sort?
Again the procedure is basically you need to recover the return address and dispose of the stack. It is not dispose the stack but dispose of the activation record. So we are recovering the return address from 168 offset. While creating activation record we said stack pointer equal to stack pointer minus 184 now we just do plus 84 and recover.

(Refer Slide Time: 39:29)

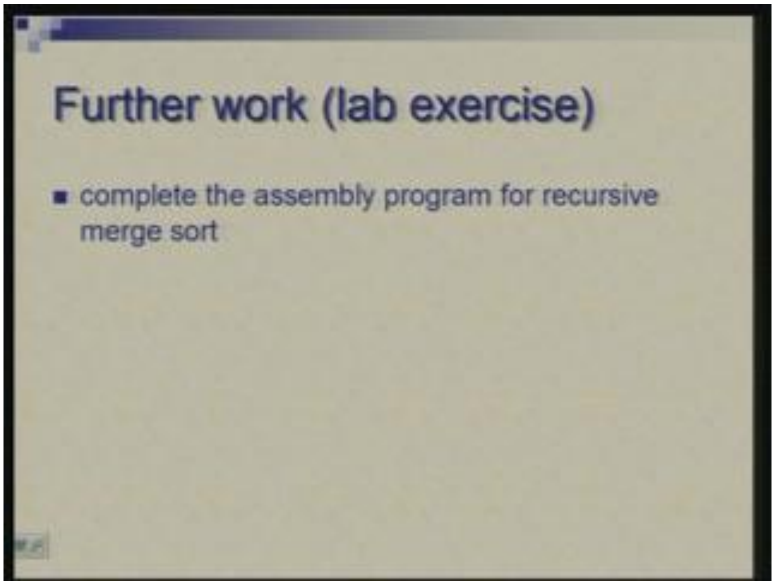


Return from sort

A'
B'
n'
A1
A2
n1
n2
return addr
A
B
n

```
lw $ra, 168($sp)
addi $sp, $sp, 184
jr $ra
```

(Refer Slide Time: 39:48)



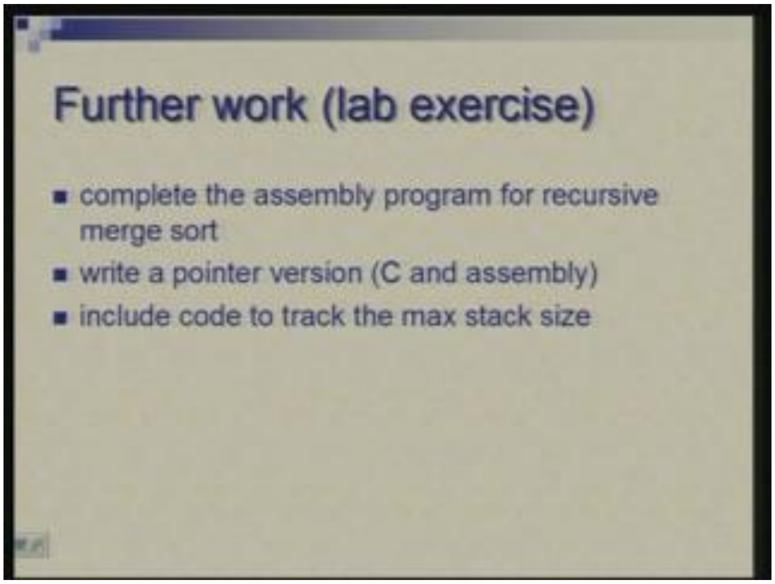
Further work (lab exercise)

- complete the assembly program for recursive merge sort

This was some portions of this exercise which I have done and I think the others can be now easily filled up by you. You should complete this assembly encoding of this exercise as part of your lab work. So you complete this, write also the main function so that you can test it and test it for different values of small m small m was the array size.

The way I have written is I have used indices but in one of the lectures you recall that we discussed how you can make the code better by using pointers or addresses. So you write a pointer version of this, first write in C, write same thing in pointers and then write corresponding assembly version.

(Refer Slide Time: 40:35)



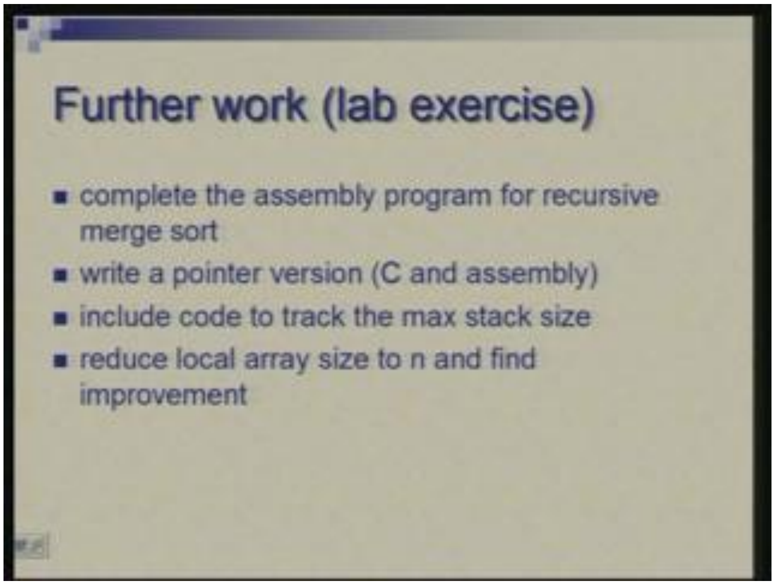
Further work (lab exercise)

- complete the assembly program for recursive merge sort
- write a pointer version (C and assembly)
- include code to track the max stack size

Now, as you would have noticed that every time a call to sort is made we were creating a large activation record of size 184; this was with the assumption that m equal to 20. so basically the area we are allocating is essentially two times m in terms of words or you could say two times m words or 8 m bytes plus a few more bytes for rest of the stuff. So now, depending upon the depth of recursive calls you will have storage on the stack growing on shrinking and we would like to find out how much the stack grows to at its peak. As you go deeper in the chain of calls it grows; when you return it shrinks
so depending upon the array size you will go up to certain depth and based on that you will have maximum usage of stack at some point of time.

You can calculate it analytically, you can also find experimentally. for analytical calculation you would need to relate it to the depth of recursion related to the size of the array and you can come up with the expression which will describe the maximum stack size used in terms of the array size as a function of m. you can also check it experimentally by introducing some additional codes which will try to keep track of the maximum value or let us say in this case since the stack grows to the lower address the minimum value of the stack pointer; you start with an empty stack assign some base address to the stack pointer and as stack grows the value decreases so you can update a value internally whenever the stack grows and you can keep track of the maximum size.

(Refer Slide Time: 42:54)

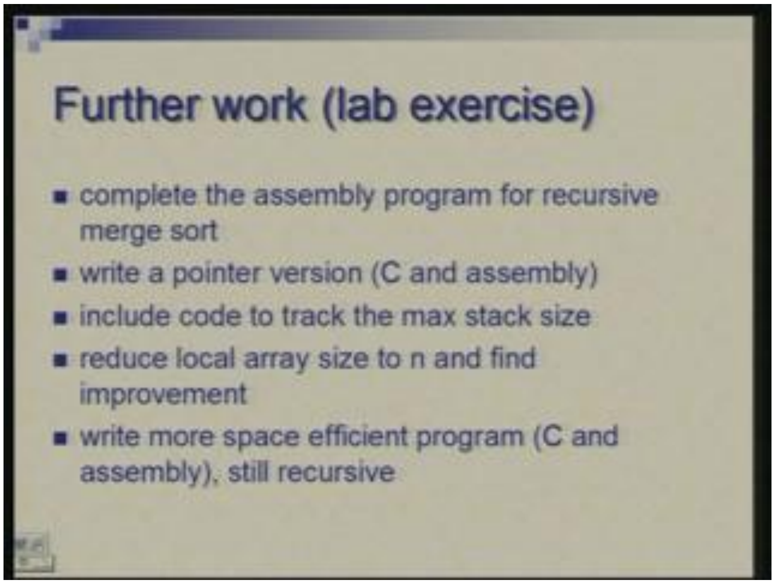


Further work (lab exercise)

- complete the assembly program for recursive merge sort
- write a pointer version (C and assembly)
- include code to track the max stack size
- reduce local array size to n and find improvement

Then I would like you to work on an improvement. What you can do is instead of allocating an array of size m always you allocate array of size n the actual value which is being passed on as a parameter; in assembly you can do that; instead of creating activation record of size 184 since you know the value which is being passed as parameter value of n…. here (Refer Slide Time: 43:49) for example look at this call; you are passing input array, this is the space for output and this is the size of the array so when a call occurs it is this size which you can use for creating a local array so you need not waste space and reserve space as much as needed and see how much improvement is required. Again you can see analytically, you can also see experimentally.

(Refer Slide Time: 44:18)



Further work (lab exercise)

- complete the assembly program for recursive merge sort
- write a pointer version (C and assembly)
- include code to track the max stack size
- reduce local array size to n and find improvement
- write more space efficient program (C and assembly), still recursive

Finally you can rewrite this function so that you do not need to create so many arrays you can restructure and it is possible that the space you use is only proportional to the data size the way we have the present algorithm written the space usage is more than proportional as you will see analytically and experimentally. But you can rewrite such that you do not create array every time you make a call because here depending upon the depth of the calls the space gets multiplied. But I would like you to rewrite first C and then assembly where the space usage is proportional to the data size and do still recursively. That is the exercise I like you to carry out.

I will stop at this point. So basically just to summarize we have seen that recursive functions entail usage of activation record on top of the stack and crucial decisions are deciding a format or the structure of activation record. Once you have decided the structure the usage becomes very easy you know what data is where although offsets are constants so you can with constants offset you can easily access data which is on the stack and then what remains is that at the time of call you appropriately create space on top of the stack, fill up the values and at the time of return recover the return address and dispose of the activation record. That is all for today.